Using Streaming SIMD Extensions to Find the Maximum/Minimum Element of a Single-Precision Floating-point Vector and its Corresponding Index

Version 1.2

01/99

Order Number: 243639-002

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Pentium® II processors, Deschutes processors, and Pentium® III processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Third-party brands and names are the property of their respective owners.

Copyright © Intel Corporation 1998, 1999

Table of Contents

Revision History

Revision	Revision History	Date
1.2	FCS version	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. *An Introduction to Hidden Markov Models*, Rabiner, L. R. and Juang, B.H., IEEE ASSP Magazine January 1986, page 5.
2. *Speech Recognition: No Longer a Dream but Still a Challenge*, Quinnel, Richard A., EDN, January 19, 1995, pgs 41 – 46.

1 Introduction

Streaming SIMD Extensions for the Intel® Architecture (IA) instruction set provide single-precision floating-point single-instruction, multiple-data (SIMD) instructions. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, and spatial (3D) audio. This application note describes how Streaming SIMD Extensions can be used to find the maximum or the minimum floating-point element of a vector and the element's corresponding index. This application note also includes examples of code that exploit the Streaming SIMD Extensions.

2 The Maximum/Minimum Algorithm

The Min/Max algorithm can be used in speech recognition. Finding the maximum or the minimum floating-point element in a vector (an array) is a very important step in the Viterbi algorithm. The Viterbi algorithm can be used in speech recognition to find the best sequence of states in a model that best describes a speech sample(refer to Section 2.1).

Three parameters are passed into this function:

- A pointer to a vector whose max/min element is to be found
- The number of elements in the vector
- A pointer of type integer that is used to store the index of the max/min element

The function returns the value of the vector's max/min floating-point element. If two or more elements have the same max/min value, the stored index represents the index of the first max/min element (the lowest index of the indices corresponding to the max/min elements).

The function can accept a vector with any 4-byte alignment (see Figure 1). However, since the Pentium® III registers (xxm floating-point registers) can hold up to 16 bytes (four floating-point elements) at a time, the vector should be aligned on a 16-byte boundary for optimal performance. If the vector is not properly aligned, the algorithm still works correctly with a slight performance penalty. For similar reasons, the best performance is achieved when the vector is also aligned at the end of the array, meaning there are no elements at the end of the array which are not part of a 16-byte aligned group of four elements. In this document and in the code, the unaligned elements at the end of the array are referred to as extra elements.

The Maximum and Minimum algorithms are very similar. The discussion from here on focuses on the maximum algorithm and highlights only the differences that occur in the minimum algorithm.

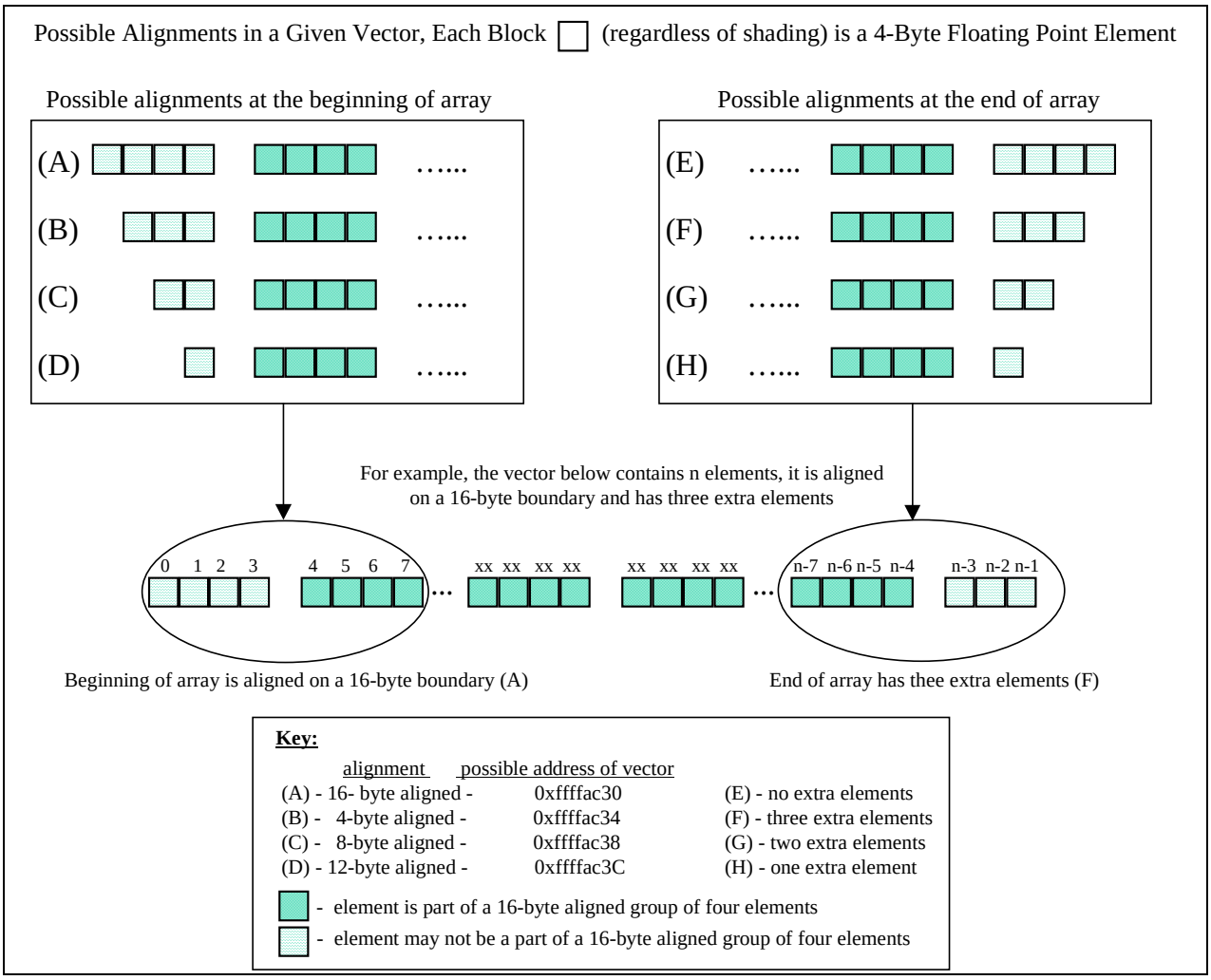

Figure 1: The different possible 4-byte alignments in a given vector

The maximum algorithm must be able to handle vectors that are unaligned at the beginning of the array and unaligned at the end of the array. For this reason, the algorithm of this function has four parts (refer to Figure 2):

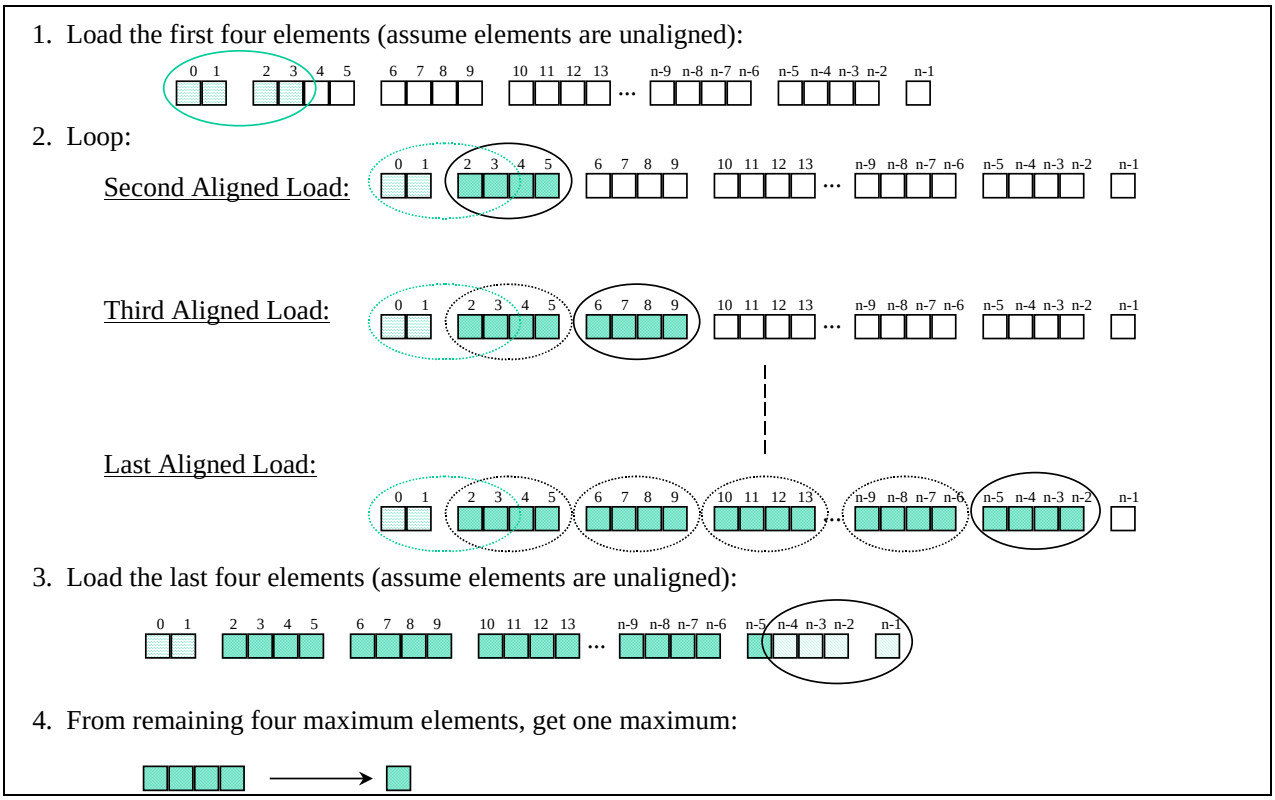

Figure 2: Algorithm of the Max Function

1. Load the first four elements into the first xmm register. Assume the elements are UNALIGNED and are the new maximum four elements. Set a second register to hold the new maximum four element's indices. For example, Step 1 of Figure 2 loads the first four elements of a vector. The indices of these first four elements are shown to be 0,1,2, and 3. The vector in this example has *n* elements, is 8-byte aligned, and has one extra element at the end.

2. Loop to get the maximum four aligned elements:

   - Load the next four ALIGNED elements into a third register.
   - Compare these current four elements (third register) with the maximum four elements (first register). Store the resulting maximum four elements as the new maximum four elements (first register).
   - Get the corresponding indices of the new maximum four elements and store them as the new maximum four element's indices (second register).

   Repeat the loop by loading the next four ALIGNED elements for the next comparison. The first register is used to hold the four maximum elements of the vector. Each of the individual maximum elements in the first register is the maximum of all elements that have occupied the corresponding register field of both the first and third registers. The loop ends when either of the following conditions is met:

- Only the extra elements remain to be loaded and compared. Extra elements are elements that are not part of the last aligned group of four elements – the vector in Figure 2 has one extra element.
- If the vector has no extra elements, end the loop when the last four aligned elements remain to be loaded and compared.

Note, if the vector's first four elements in the vector are unaligned then the second aligned load will overlap the first load (see the SECOND ALIGNED LOAD in Figure 2).

3. Next, after the loop is completed, load the last four elements of the vector into the third register. Assume the last four elements are UNALIGNED. If the elements at the end of the array are unaligned, the last four elements will overlap with the previous load (see Step 3 in Figure 2). Compare these last four elements (third register) with the four maximum elements (first register), and if necessary update the first register to hold the new maximum four elements. Then, if necessary, update the corresponding indices (second register) of the new maximum elements.
4. From the four maximum elements and their indices, shuffle and compare the four maximum elements to find one maximum of the four and its corresponding index. Return the maximum element and use the pointer (passed into the function) to store the index. Figures 3, 4, and 5 describe Step 4 using an example.

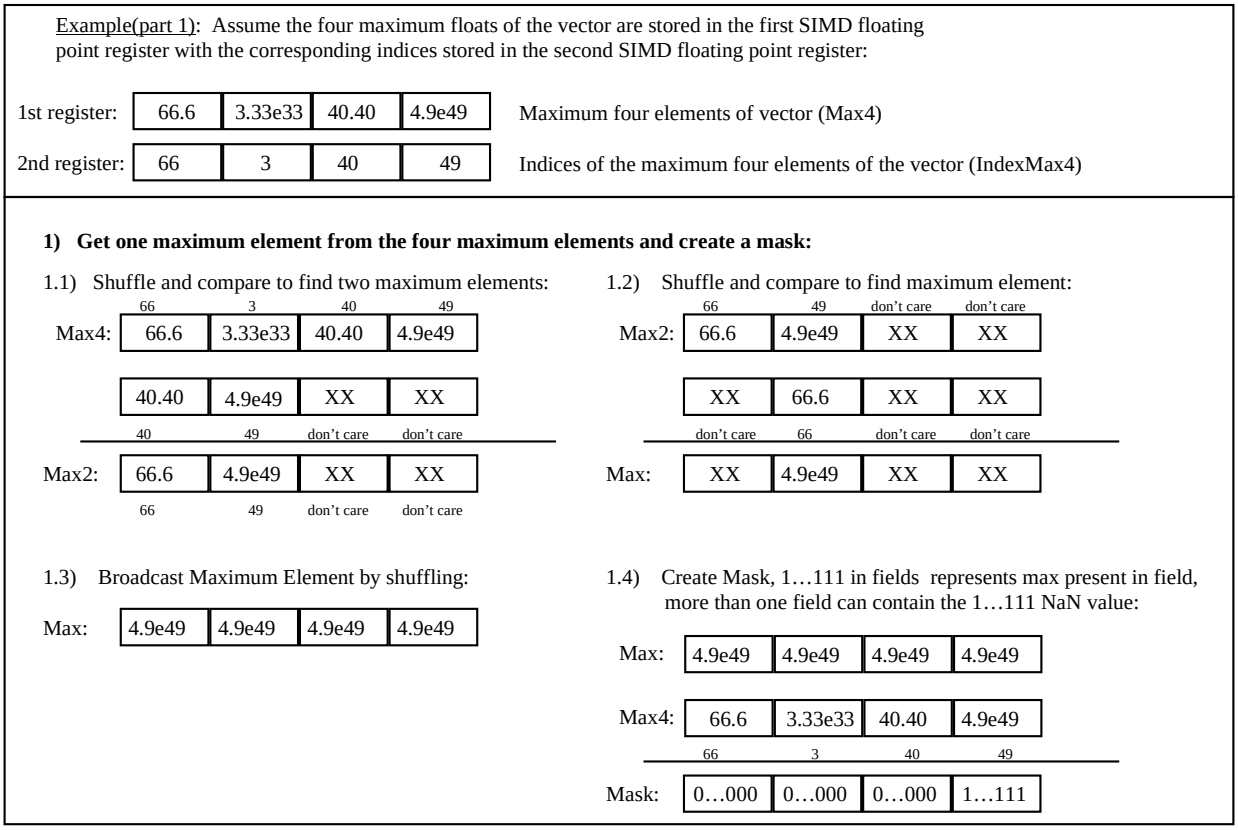

Figure 3: Step 4 of the max algorithm – get maximum element and a mask.

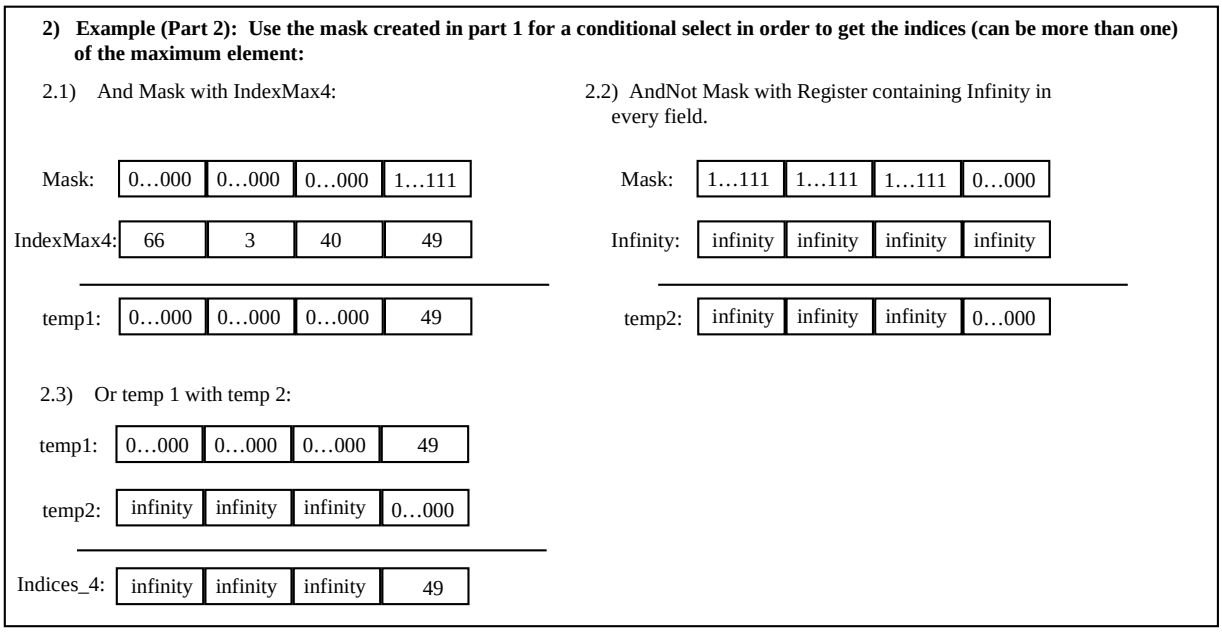

Figure 4: Step 4 of the max algorithm – get corresponding indices of the maximum element.

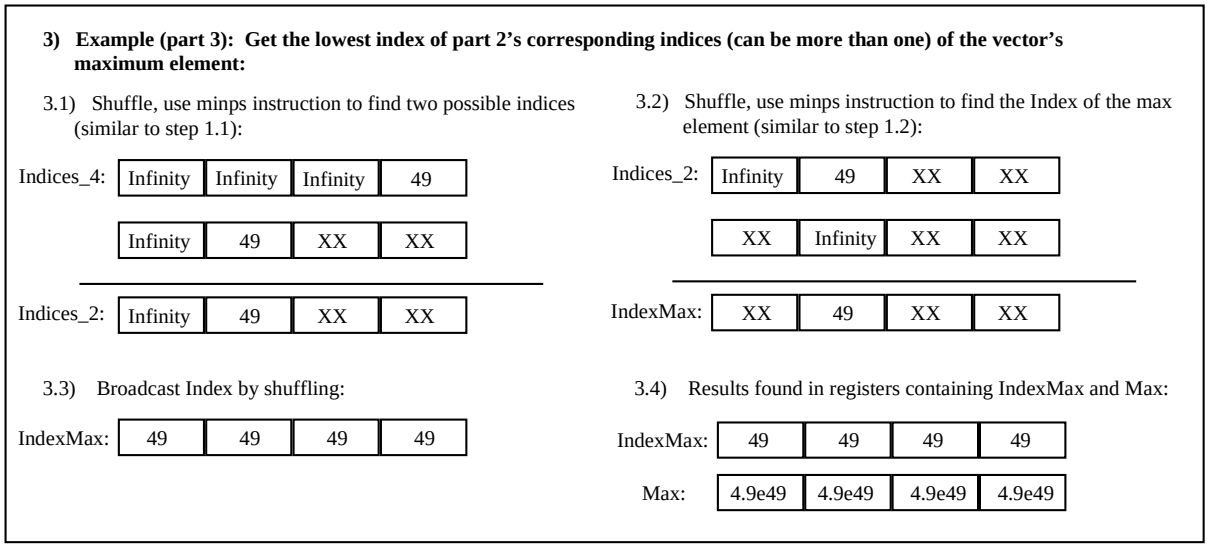

Figure 5: Step 4 of the max algorithm – get lowest index of maximum element.

2.1 Applications for the Maximum Algorithm

The maximum algorithm is especially useful in the Viterbi algorithm. One use of the Viterbi algorithm is in evaluating Hidden Markov Models (HMM) for speech recognition. For a definition of an HMM (symbolized as λ), and how the Viterbi algorithm evaluates an HMM see [1].

A sequence of sound spectra generated by an utterance (words or syllables) can be modeled through a set of state movements [2]. Given a speech sample represented by an observation vector O, one can compute the probability that the observation sequence can be generated by a set of state sequences in a

given HMM, $P(O|\lambda)$. The model may generate the same observation sequence through different state sequences, but each state sequence is associated with a different probability. The state sequence with the highest probability is the path that is most likely to have generated the observation sequence. This state sequence is known as the best path (P^V):

$P^V = \max[P(Q|O,\lambda)]$, where Q is a state sequence that can generate O (λ is the HMM model)

$P(O|\lambda) \approx P^V$

To do an exhaustive search on an HMM for the best path is impractical. Instead, Viterbi decoding can provide an efficient algorithm to find the best path. This algorithm can be used to find both the best path and the state sequence of the best path.

The Viterbi algorithm has three steps [1]. The first two steps find the possible paths that can generate the observation sequence and stores the probability per path in a vector. For the third step, the user can use a maximum algorithm to find the best path (the path that best represents the given speech sample). The maximum function, as implemented in this application note, provides the functionality required for finding and identifying the best path.

2.2 Implementing the Maximum Algorithm

The algorithm makes three assumptions:

- The vector passed into the function is aligned on a 4-byte boundary – the size of a floating-point element (refer to Figure 1).
- The vector contains less than 2^{23} - 1 elements.
- Finding a new maximum element in the vector is a rare event.

The first assumption was discussed above. The assumption that there are less than 2^{23} - 1 elements present in the vector is not much of a restriction since an array with about 8.4 million elements can be supported. The limit on the number of elements is due to using the xmm registers to hold the indices of the vector. The xmm registers support 23 precision bits for each floating-point element. The algorithm also assumes that finding a new maximum element in the vector is a rare event (refer to Section 2.2.1). If the data meets this last assumption the function will provide optimal performance. If the data does not meet this assumption, see Section 3.2 to optimize the function for this type of data.

2.2.1 Techniques

Three optimization techniques were implemented to increase the function's performance. First, the loop of this function contains instructions that are resource dependent. This dependency causes the instructions to execute sequentially. Executing sequentially does not take advantage of the Streaming SIMD Extensions Out-of-Order Core. To reduce the resource dependency, a branch was placed inside the loop. The branch increased the number of instructions in the loop, but by assuming that very few maximums will be found, the three instructions used to find the maximum four elements and their indices rarely will be executed and more instructions are able to execute in parallel. During testing, this optimization reduced the execution time by approximately 20%.

For the second optimization, the `maxps` instruction was used inside the loop of the max function to find the index of the new maximums. Usually, to avoid branches, the following instructions are used to create a conditional select (see Figure 4 for an example of a conditional select):

1. Compare to get a Mask

2. AND Mask with the first possible solution
3. ANDNOT Mask with the second possible solution
4. OR the result of the AND and ANDNOT to get the final result

But, in this case, because the algorithm starts comparing elements from the beginning of the vector (index 0) to the end of the vector (index *n*-1 of a vector with *n* elements), the indices of the elements keep increasing as the next set of elements are compared. Therefore, a new maximum element will have an index with a greater value than the index of the previous maximum element. For this reason, one instruction from the conditional select can be eliminated to find the indices of the new maximum elements (refer to Figure 6):

1. Compare to get a Mask
2. AND Mask with indices of the next elements being compared
3. Use MAXPS: get the maximum of the previous indices and new indices

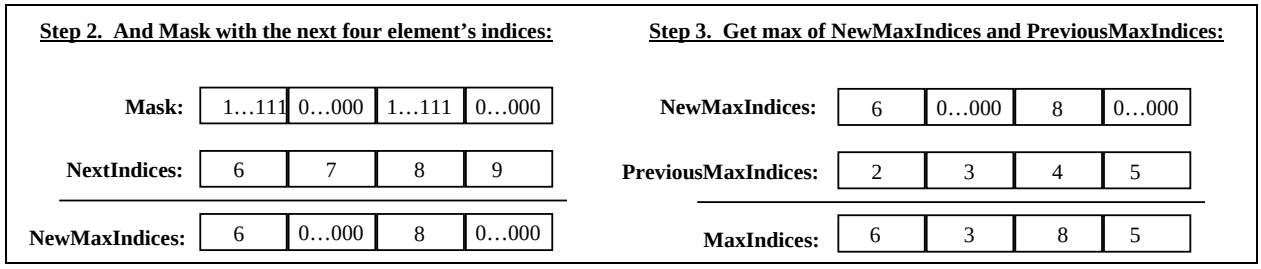

Figure 6: Steps to get the indices of the max elements (Optimized Conditional Select)

Finally, for the third optimization that was implemented, the `minps` instruction was used to find the index of the first maximum element of the vector. That is, if two or more elements of the vector have the same value and these elements are the maximum element of the vector, then the algorithm stores the index with the lowest value – the first occurrence of the maximum element. Therefore, once all the elements have been compared and only four maximum values remain in one register, the first occurrence of the maximum element is found as follows (see Figure 5 for an example):

1 Get the max element of the remaining four max elements

2 Compare the max element with the previous four max elements to get a mask describing which fields of the register hold the max element

3 AND this mask with the indices of the previous four max elements

4 Place infinity into the fields containing zeroes.

5 Use the `minps` *instruction to get the index with the lowest value - the first occurrence of the maximum element*

The loop of this algorithm has very few instructions. The small size of the loop means that loop unrolling and load forwarding cannot increase the performance, because these techniques are already being done by the processor's Out-of-Order Core. Tests reported that loop unrolling and load forwarding actually added more cycles compared to the loop that did not implement these software optimization techniques. If the loop employed more instructions (about twenty), loop unrolling and forwarding the loads could be helpful.

The prefetching instruction was not implemented for two reasons:

- As discussed in Section 2.1 this algorithm will probably be used in the Viterbi algorithm. In this case, the elements of the vector are assumed to be in cache and prefetching is unnecessary.
- The loop in this algorithm has very few instructions, therefore, as discussed above, the processor's Out-of-Order Core will already forward the load.

3 Performance Considerations

Considerations

If the data has many new maximums the user may want to remove the branch from the max function by substituting the loop with the following code (see Section 2.2.1 for a discussion on the branch):

Example 1: Loop without branch

```
// xmm0:     current max values
// xmm1:     mask generated by comparison
// xmm2:     indices of current max values
// xmm3:     indices of next four floats
// xmm4:     holds the number 4.0 in each field of the register
// esi+ecx:  address of next four aligned floats in array

 loopMax:
    cmpnleps   xmm1, xmm0      ; ** each 111...1 field signals a new max found
    maxps      xmm0, [esi+ecx] ; ** Place the maximum four values into xmm0
    andps      xmm1, xmm3      ; get corresponding indices for new max values
    add        ecx,  16        ; increment pointer to point to the next 4 floats
    maxps      xmm2, xmm1      ; update indices of the 4 max values
    addps      xmm3, xmm4      ; get corresponding indices of next four floats
    movaps     xmm1, [esi+ecx] ; Load next four aligned floats
    jnz        loopMax
```

Similarly, for the minimum version of this function, if the data has many new minimums the above code can be used with a few modifications. The two instructions that will change are marked with ** in the comments. Replace the marked instructions with the following instructions: `cmpltps`, `minps`.

The source code in Section 6 contains comments that describe what section can be replaced by the code described in example 1.

4 Conclusion

The max/min functions implemented using Streaming SIMD Extensions are faster than the C-implementation.. This speedup is due to the SIMD floating-point instructions, especially the `maxps` and `minps` instructions found in the Streaming SIMD Extensions instruction set.

5 C Coding Example

The algorithm for the max function implemented in C is shown below. The algorithm for the min function is not shown since it is very similar (change sign from '<' to '>' in condition, marked as **).

```
/* C_Max.cpp - Baseline code for Max function
 */

float Max_C(float *src,    // Pointer to vector whose maximum-valued element
                           // is to be found
            int n,         // Number of elements to be operated on
            int *position) // Index of maximum-valued element
{
  float max_float = src[0];
  int   max_index = 0;
  int   i;

  for( i=1; i<n; i++)
  {
    if( max_float < src[i])   //** If new max found, store value and index
    {
      max_float  = src[i];
      max_index = i;
    }
  }
  *position = max_index;
  return(max_float);
}
```

6 Streaming SIMD Extensions Assembly Code Example

This section contains the assembly implementation of the max algorithm. The min algorithm is similar:

The only changes in the min function are as follows:

- The `minps` instruction replaces all but two `maxps` instructions in finding the minimum 4 floats
- The `cmpltps` instruction replaces the two `cmpnleps` when creating a mask for the indices
- The C code (the code that handles the case when there are 12 or less elements) in the function that finds the minimum should change its condition from '<' to '>':

Code from Max Function

```
if(maxFloat < src[i])
{
    maxFloat = src[i];
    maxIndex = i;
}
```


Code Converted for Min Function

```
if(minFloat > src[i])
{
    minFloat = src[i];
    minIndex = i;
}
```

?

Note that the following instructions are the same in both the max and min functions (see Section 2.2.1 for a discussion on these optimizations):

- Two `maxps` instructions used to find the indices of the four max/min elements (do not change these instructions to `minps` – marked as $$)
- Two `minps` instructions used to find the index of the first max/min element (do not change these instructions to `maxps` – marked as ##)

```
////////////////////////////////////////////////////////////////////////////////
// Max_XMM: Returns the maximum valued element and stores its index in the variable     //
// -------  'position'.  If the maximum value is represented by two or more             //
//          elements in the array the index will represent the first maximum value      //
//          encountered.                                                                //
////////////////////////////////////////////////////////////////////////////////
//                                                                                     //
// Assumption:                                                                         //
// ===========          o Vector is aligned on a 4-byte boundary                       //
//                                                                                     //
// Algorithm:  A pointer to a vector with elements of type floating-point is passed    //
// ---------   into this function.  The vector may have one, two, or three elements    //
//             that are not aligned on a 16-byte boundary as shown below:              //
//                                                                                     //
//             Aligned on a 16-byte boundary:            xxxx xxxx xxxx ...            //
//             Aligned on a 12-byte boundary:               x xxxx xxxx ...            //
//             Aligned on an 8-byte boundary:              xx xxxx xxxx ...            //
//             Aligned on a  4-byte boundary:             xxx xxxx xxxx ...            //
//                                                                                     //
//         (note: 'x' refers to a floating-point element in the vector, the            //
//                   elements are grouped on 16-byte boundaries)                       //
//                                                                                     //
```

```
//            The first four elements in the vector will be assumed unaligned and      //
//            will be loaded with the movups instruction.  The last four elements      //
//            in the vector must also be assumed to be unaligned and will also be      //
//            loaded with the movups instruction.  There can be one, two, or three     //
//            extra elements that are not aligned at the end of the vector as          //
//            shown below:                                                             //
//                                                                                     //
//            One   extra element  at the end of the vector: ... xxxx xxxx x           //
//            Two   extra elements at the end of the vector: ... xxxx xxxx xx          //
//            Three extra elements at the end of the vector: ... xxxx xxxx xxx         //
//                                                                                     //
//            For this reason, the algorithm of this function has four parts:          //
//                                                                                     //
//            *** Part 1:  Load the first four elements, assume the elements are       //
//                         UNALIGNED and are the new maximum four elements.            //
//                                                                                     //
//            *** Part 2:  Loop: Load the next four ALIGNED elements, compare the      //
//                         current four elements with the previous four maximum        //
//                         elements and store the four new maximum elements.  Get      //
//                         the corresponding indices of the new maximum elements.      //
//                         If the data is not properly aligned and extra elements      //
//                         are present, loop until only the extra elements remain      //
//                         to be compared.  If the data is properly aligned, the       //
//                         loop will end only when the last four elements remain       //
//                         to be compared.                                             //
//                                                                                     //
//            *** Part 3:  Load the last four elements, assume the elements are        //
//                         UNALIGNED.  Compare the last four elements with the         //
//                         previous four maximum elements and store the four new       //
//                         maximum elements.  Get the corresponding indices of the     //
//                         four new maximum elements.                                  //
//                                                                                     //
//            *** Part 4:  From the four maximum elements, get one maximum and its     //
//                         corresponding index.  Return the maximum element and        //
//                         store the index.                                            //
/////////////////////////////////////////////////////////////////////////////////////////



#include <assert.h>

// firstIndices  - initializes reg holding the indices of the first four elements
// const_4_4_4_4 - initializes reg used to increment the next indices
// infinityArray - used so the minps instruction can be used to find the index of
//                 the first maximum element.
static const unsigned int infinity          = 0x07F800000;
static const float firstIndices[4]          = {0.0f,1.0f,2.0f,3.0f};
static const float const_4_4_4_4[4]         = {4.0f,4.0f,4.0f,4.0f};
static const unsigned int infinityArray[4]  = {infinity,infinity,infinity,infinity};
```

```
float Max_XMM(float *src,      // Pointer to vector whose maximum-valued element
                               //   is to be found
           int n,              // Number of elements to be operated on
           int *position)      // Index of maximum-valued element
{
// Assertions
   // Vector is 4-byte aligned
   assert(((unsigned int) src  & 0x03) == 0);

   // Number of elements in vector > ((2^23)-1), ((2^23)-1) is the largest signed
   // integer represented by the xmm register precision bits
   assert(n <= 8388607);

// Use C Code to handle cases when vector contains 12 or less elements
   if (n<=12)
   {
      int   maxIndex = 0;
      float maxFloat = src[0];
      int i;

      for(i=1; i<n; i++)
      {
         if(maxFloat < src[i])
         {
            maxFloat = src[i];
            maxIndex = i;
         }
      }
      *position = maxIndex;
      return(maxFloat);
   }

// Variable Declarations

   float maxFloat;      // will be used to store maximum element
   int   maxIndex;      // will be used to store index of maximum element

   // possible values of aligned: 1 -> signifies vector is 12 byte aligned
   //                    =======  2 -> signifies vector is  8 byte aligned
   //                             3 -> signifies vector is  4 byte aligned
   //                             4 -> signifies vector is 16 byte aligned
   int aligned = 4 - (((unsigned int) src >> 2) & 0x03);

   // alignedPS initializes the ecx register for loop
   // alignedPS  = ((elements in vector aligned on 16-byte boundary) * 4)
   int alignedPS = ((n-aligned-1) &~ (0x03))<<2;

   // last_four points to the last four elements in the vector
```

```
   float *lastFour = (src+n-4);

   // nextIndices - initializes reg holding indices of the next aligned elements
   // lastIndices - initializes reg holding the indices of the last four elements

   float nextIndices[4] = {(float) aligned,       (float) aligned+1.0f,
                           (float) aligned+2.0f,    (float) aligned+3.0f};
   float lastIndices[4] = {(float)n-4.0f, (float)n-3.0f,(float)n-2.0f, (float)n-1.0f};

   // aligned is used to increment pointer to point to the next aligned four elements
   aligned = aligned << 2;

   __asm
   {
      // copy parameters to IA registers
      mov      esi, src
      mov      ecx, alignedPS

      // initialize index & constant registers, assume unaligned
      movups  xmm2, firstIndices
      movups  xmm3, nextIndices
      movups  xmm4, const_4_4_4_4

      // Get first four elements
      movups   xmm0, [esi]    ; Load first four floats, assume unaligned
      add      esi, aligned   ; pts to the next four 16 byte aligned floats in array
      add      esi, ecx       ; esi now pts to the last four aligned floats
      neg      ecx            ; [esi + ecx] will hold the next four floats

      // loop to get next four elements, compare to find the maximum four elements, loop
      // until only the last four elements remain to be compared.
      // loop assumes very few maximums will be found


      movaps   xmm1, [esi+ecx]   ; Load next four aligned floats
/* If the data has many new maximums the user may want to remove the branch (maxFound) from the
loop below by substituting the entire loop below with the code described in example 1 (see
Section 2.2.1 for a discussion on the branch) */
//Start possible code replacement with code of example 1
loopMax:
      cmpnleps xmm1, xmm0     ; each 111...1 field signals a new maximum found
      movmskps eax, xmm1      ; if new max found, get max and index
      cmp      eax, 0
      je       noMax
maxFound:
      maxps   xmm0, [esi+ecx] ; Place the maximum four values into xmm0
      andps   xmm1, xmm3      ; get corresponding indices for the new maximum values
      maxps   xmm2, xmm1      ; $$ update register holding the indices of the 4 max values
noMax:
      add     ecx,  16        ; increment pointer to point to the next four floats
      addps   xmm3, xmm4      ; increment indices to correspond to the next four floats
```

```
      movaps  xmm1, [esi+ecx] ; Load next four aligned floats
      jnz     loopMax
// End of possible code replacement

      // Get last four elements, find the maximum four elements and their indices
      mov     esi,  lastFour  ; point to last four elements
      movups  xmm3,lastIndices; load the last four indices, assume unaligned
      movups  xmm1, [esi]     ; load the last four elements, assume unaligned
      movaps  xmm5, xmm0      ; copy maximum 4 values for comparison
      maxps   xmm0, xmm1      ; Place the maximum four values into xmm0
      cmpnleps xmm1, xmm5     ; each 111...1 field signals a new maximum found
      andps   xmm1, xmm3      ; get corresponding indices for the new maximum values
      maxps   xmm2, xmm1      ; $$ update register holding the indices of the 4 max values

   // Get the maximum float of the remaining 4 floats with its corresponding index.
   // If the maximum value is represented by two or more elements in the array
   // get the index representing the first maximum value that was encountered

      movups  xmm1, infinityArray  ; xmm1 = [infinty, infinity, infinity, infinity]
                                   ; xmm0 = [m3   m2   m1  m0]
      shufps   xmm5, xmm0, 0x40    ; xmm5 = [m1   m0   xx  xx]
      maxps    xmm5, xmm0          ; xmm5 = [max2 max1 xx  xx]
      shufps   xmm7, xmm5, 0x30    ; xmm7 = [xx   max2 xx  xx]
      maxps    xmm5, xmm7          ; xmm5 = [xx   max  xx  xx]
      shufps   xmm5, xmm5, 0xAA    ; xmm5 = [max  max  max max]

      cmpeqps  xmm0, xmm5          ; each 111...1 field signals max present in field

      andps    xmm2, xmm0          ; get corresponding indices of the maximum values
      andnps   xmm0, xmm1          ; place infinity in fields containing 000...0's
      orps     xmm2, xmm0          ; xmm2 holds either indices of max elements or infinity

                                   ; xmm2 = [i3     i2     i1    i0]
      shufps   xmm6, xmm2, 0x40    ; xmm6 = [i1     i0     xx    xx]
      minps    xmm2, xmm6        ; ## xmm2 = [index2 index1 xx    xx]
      shufps   xmm7, xmm2, 0x30    ; xmm7 = [xx     index2 xx    xx]
      minps    xmm2, xmm7        ; ## xmm5 = [xx     index  xx    xx]
      shufps   xmm2, xmm2, 0xAA    ; xmm5 = [index  index  index index]

      // Store results
      cvtss2si eax,        xmm2
      mov      maxIndex,   eax
      movss    maxFloat,   xmm5
   }
      // Store and return results
      *position = maxIndex;
      return(maxFloat);
}
```

7 Code Example Using Compiler Intrinsics for Streaming SIMD Extensions

Below is the intrinsics implementation of the max algorithm, the min algorithm is similar:

```
// Max_Intr: Intrinsics implementation of max kernel
//************************************************

#include "xmmintrin.h"
#include <assert.h>


// const_4s contains [4,4,4,4], it increments indexCurr4 to the next indices
static const __m128 const_4s  = _mm_set_ps1(4.0f);
static const int   i_infinity = 0x07F800000;
static const float f_infinity = *(float*)&i_infinity;

float Max_Intr(float *src, int n, int *position)
{


// Assertions
   // Vector is 4-byte aligned
   assert(((unsigned int) src  & 0x03) == 0);

   // Number of elements in vector > ((2^23)-1), ((2^23)-1) is the largest signed
   // integer represented by the simd fp register precision bits
   assert(n <= 8388607);


   float maxFloat = src[0];
   int   maxIndex = 0;

   // If a parameter into a function is a pointer,init the pointer during engineering releases
   *position = maxIndex;


// Use C++ Code to handle cases when vector contains twelve or less elements
   if (n <= 12)
   {
      for(int i=1; i<n; i++)
      {
         if(maxFloat < src[i])
         {
            maxFloat = src[i];
            maxIndex = i;
         }
```

```
      }
      *position = maxIndex;
      return(maxFloat);
   }

// Variable Declarations

   // possible values of aligned: 1 -> signifies vector is 12 byte aligned
   //                    =======  2 -> signifies vector is  8 byte aligned
   //                             3 -> signifies vector is  4 byte aligned
   //                             4 -> signifies vector is 16 byte aligned
   int aligned = 4 - (((unsigned int) src >> 2) & 0x03);

   // numLoops = n - (unaligned elements at the beginning of the vector)
   int numLoops = n - aligned;

   // lastFour points to the last four elements in the vector
   float *lastFour = (src+n-4);

   int maskBits;  // after a simd fp comparison, maskBits gets a 4 bit mask formed by the
                  // most sig. bits of the 4 FP fields to describe the comparison results
   __m128 max4,   // max4 holds the four maximum floats
                  // curr4 holds the next four aligned elements
         curr4       = _mm_set_ps((float)src[aligned+3], (float)src[aligned+2],
                                  (float)src[aligned+1], (float)src[aligned]),
         mask,    // mask holds the result of comparison between max4 and curr4
                  // indexMax4 holds the indices of the maximum floats in max4
         indexMax4   = _mm_set_ps(3.0f,2.0f,1.0f,0.0f),
                  // indexCurr4 holds the indices of the floats in curr4
         indexCurr4  = _mm_set_ps((float)aligned+3.0f, (float)aligned+2.0f,
                                  (float)aligned+1.0f, (float)aligned),
                  // lastIndices holds the indices of the last four elements
         lastIndices = _mm_set_ps((float)n-1.0f, (float)n-2.0f,(float)n-3.0f, (float)n-4.0f),
                  // infinity holds 4 bit patterns used to init a fp infinity per field
         infinity    = _mm_set_ps(f_infinity, f_infinity, f_infinity, f_infinity);


   // Load first 4 floats, assume elements are unaligned
   max4 = _mm_loadu_ps(src);

   // loop to get next four elements, compare to find the maximum four elements, loop
   // until only the extra elements or the last four elements remain to be compared.
   for(int nextFour = 4; nextFour < numLoops ; nextFour+=4)
   {
      mask = _mm_cmpnle_ps(curr4,max4);
      maskBits = _mm_movemask_ps(mask);   // get 4 bit mask
   // If a mask was created, a new max is found, else get the next 4 elements
      if (maskBits > 0)
      {
```

```
            max4 = _mm_max_ps(max4,curr4);         // update max 4 elements
            mask = _mm_and_ps(indexCurr4,mask);    // get new indices
            indexMax4 = _mm_max_ps(indexMax4,mask); // update indices of max 4 elements
         }
      // get the next 4 elements
         indexCurr4 = _mm_add_ps(indexCurr4,const_4s);// increment indices of next 4 elems
         curr4 = _mm_load_ps(src + aligned + nextFour);// load the next 4 elements
      }

      // Get last 4 elements which can include any extra elements, assume elements are unaligned
      // Find the maximum four elements and their indices
      indexCurr4 = lastIndices;
      curr4 = _mm_loadu_ps(lastFour);
      mask  = _mm_cmpnle_ps(curr4,max4);
      max4  = _mm_max_ps(max4,curr4);
      mask  = _mm_and_ps(indexCurr4,mask);
      indexMax4 = _mm_max_ps(indexMax4,mask);

      // Get the maximum element with its corresponding index
      mask  = max4;
                                              // max4  = [m3   m2   m1  m0]
      curr4 = _mm_shuffle_ps(curr4,max4,0x40);  // curr4 = [m1   m0   xx  xx]
      max4  = _mm_max_ps(curr4,max4);           // max4  = [max2 max1 xx  xx]
      curr4 = _mm_shuffle_ps(curr4,max4,0x30);  // curr4 = [xx   max2 xx  xx]
      max4  = _mm_max_ps(curr4,max4);           // max4  = [xx   max  xx  xx]
      max4  = _mm_shuffle_ps(max4,max4,0xAA);   // max4  = [max  max  max max]

      mask  = _mm_cmpeq_ps(mask,max4);          // which fields hold max element?

      indexMax4  = _mm_and_ps(mask,indexMax4);  // mask out unneeded indices
      mask       = _mm_andnot_ps(mask,infinity);   // place infinity in place of 000...0's
      indexMax4  = _mm_or_ps(indexMax4,mask);      // indexMax4 holds index of maximum or infinity
                                                          //iMax4  = [i3   i2    i1 i0]
      indexCurr4 = _mm_shuffle_ps(indexCurr4,indexMax4,0x40); //iCurr4 = [i1   i0    xx xx]
      indexMax4  = _mm_min_ps(indexCurr4,indexMax4);         //iMax4  = [inx2 inx1  xx xx]
      indexCurr4 = _mm_shuffle_ps(indexCurr4,indexMax4,0x30); //iCurr4 = [xx   inx2  xx xx]
      indexMax4  = _mm_min_ps(indexCurr4,indexMax4);         //iMax4  = [xx   i     xx xx]
      indexMax4  = _mm_shuffle_ps(indexMax4,indexMax4,0xAA); //iMax4  = [i    i     i  i ]

      // Store and return result
      _mm_store_ss(&maxFloat,max4);
      *position  = _mm_cvt_ss2si(indexMax4);

      return(maxFloat);
   }
```

8 Streaming SIMD Extensions Fvec32 Code Example

Below is the Fvec32 implementation of the max algorithm, the min algorithm is similar:

```
// Max_Fvec32: F32vec4 C++ class implementation of max kernel
//*********************************************************


#include "fvec.h"
#include <assert.h>

// firstIndices  - initializes reg holding the indices of the first four elements
static const F32vec4 firstIndices(3.0f,2.0f,1.0f,0.0f);
static const int i_infinity = 0x07F800000;
static const float f_infinity = *(float*)&i_infinity;

float Max_Fvec32(float *src, int n, int *position)
{

// Assertions
   // Vector is 4-byte aligned
   assert(((unsigned int) src  & 0x03) == 0);

   // Number of elements in vector > ((2^23)-1), ((2^23)-1) is the largest signed
   // integer represented by the xmm register precision bits
   assert(n <= 8388607);

   float maxFloat = src[0];
   int   maxIndex = 0;

   // If a parameter into a function is a pointer,init the pointer during engineering releases
   *position = maxIndex;


// Use C++ Code to handle cases when vector contains twelve or less elements
   if (n <= 12)
   {
      for(int i=1; i<n; i++)
      {
         if(maxFloat < src[i])
         {
            maxFloat = src[i];
            maxIndex = i;
         }
      }
      *position = maxIndex;
      return(maxFloat);
   }
```

```
// Variable Declarations

   // possible values of aligned: 1 -> signifies vector is 12 byte aligned
   //                    =======  2 -> signifies vector is  8 byte aligned
   //                             3 -> signifies vector is  4 byte aligned
   //                             4 -> signifies vector is 16 byte aligned
   int aligned = 4 - (((unsigned int) src >> 2) & 0x03);

   // numLoops = n - (unaligned elements at the beginning of the vector)
   int numLoops = n - aligned;

   // lastFour points to the last four elements in the vector
   float *lastFour = (src+n-4);

   int mask4Bits; // after a simd fp comparison, maskBits gets a 4 bit mask formed by the
                  // most sig. bits of the 4 FP fields to describe the comparison results
   F32vec4 max4,  // max4 holds the four maximum floats
                  // curr4 holds the next four aligned elements
         curr4((float)src[aligned+3],  (float)src[aligned+2],
               (float)src[aligned+1],  (float)src[aligned]),
         mask,    // mask holds the result of comparison between max4 and curr4
                  // indexMax4 holds the indices of the maximum floats in max4
         indexMax4 = firstIndices,
                  // indexCurr4 holds the indices of the floats in curr4
         indexCurr4((float)aligned+3.0f, (float)aligned+2.0f,
                    (float)aligned+1.0f, (float)aligned),
                  // const_4_4_4_4 contains [4,4,4,4], it will be used to
                  // increment indexCurr4 to the next indices
         const_4_4_4_4(4.0f,4.0f,4.0f,4.0f),
                  // lastIndices holds the indices of the last four elements
         lastIndices((float)n-1.0f, (float)n-2.0f,(float)n-3.0f, (float)n-4.0f),
                  // infinity holds 4 bit patterns used to init a fp infinity per field
         infinity(f_infinity, f_infinity, f_infinity, f_infinity);

   // Load first 4 floats, assume elements are unaligned
   max4 = (F32vec4) _mm_loadu_ps(src);

   // loop to get next four elements, compare to find the maximum four elements, loop
   // until only the extra elements or the last four elements remain to be compared.
   for(int nextFour = 4; nextFour < numLoops; nextFour+=4)
   {
      mask = cmpnle(curr4,max4);
      mask4Bits = _mm_movemask_ps(mask);  // get 4 bit mask
   // If a mask was created, a new max is found, else get the next 4 elements
      if (mask4Bits > 0)
      {
         max4 = simd_max(max4,curr4);        // update max 4 elements
         mask &= indexCurr4;                 // get new indices
```

```
         indexMax4 = simd_max(indexMax4,mask);  // update indices of max 4 elements
      }
   // get the next 4 elements
      indexCurr4 += const_4_4_4_4;             // increment indices of next 4 elements
      curr4 = *(F32vec4*)(src + aligned + nextFour);// load the next 4 elements
   }

   // Get last 4 elements which can include any extra elements, assume elements are unaligned
   // Find the maximum four elements and their indices
   indexCurr4 = lastIndices;
   curr4  = (F32vec4) _mm_loadu_ps(lastFour);
   mask   = cmpnle(curr4,max4);
   max4   = simd_max(max4,curr4);
   mask  &= indexCurr4;
   indexMax4 = simd_max(indexMax4,mask);

   // Get the maximum element with its corresponding index
   mask  = max4;
                                          // max4  = [m3   m2   m1  m0]
   curr4 = _mm_shuffle_ps(curr4,max4,0x40);  // curr4 = [m1   m0   xx  xx]
   max4  = simd_max(curr4,max4);             // max4  = [max2 max1 xx  xx]
   curr4 = _mm_shuffle_ps(curr4,max4,0x30);  // curr4 = [xx   max2 xx  xx]
   max4  = simd_max(curr4,max4);             // max4  = [xx   max  xx  xx]
   max4  = _mm_shuffle_ps(max4,max4,0xAA);   // max4  = [max  max  max max]


   // indexMax4 holds index of maximum or infinity
   indexMax4 = select_eq(mask,max4,indexMax4,infinity);
                                                        //iMax4  = [i3   i2    i1 i0]
   indexCurr4 = _mm_shuffle_ps(indexCurr4,indexMax4,0x40);   //iCurr4 = [i1   i0    xx xx]
   indexMax4  = simd_min(indexCurr4,indexMax4);              //iMax4  = [inx2 inx1  xx xx]
   indexCurr4 = _mm_shuffle_ps(indexCurr4,indexMax4,0x30);   //iCurr4 = [xx   inx2  xx xx]
   indexMax4  = simd_min(indexCurr4,indexMax4);              //iMax4  = [xx   i     xx xx]
   indexMax4  = _mm_shuffle_ps(indexMax4,indexMax4,0xAA);    //iMax4  = [i    i     i  i ]


   // Store and return result
   _mm_store_ss(&maxFloat,max4);
   *position  = _mm_cvt_ss2si(indexMax4);
   return(maxFloat);
}
```